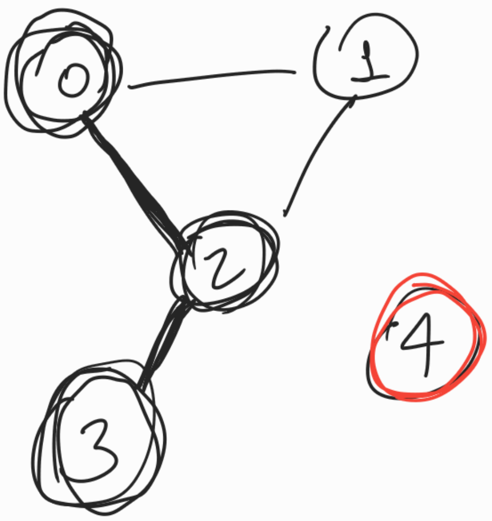
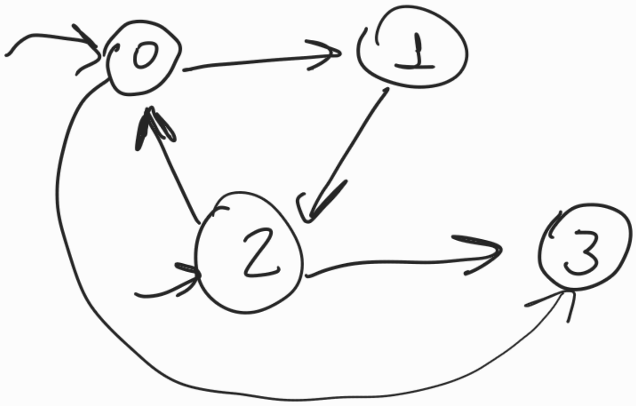
→ O que é um grafo?

↳ Um conjunto de vértices e um conjunto de arestas

Vértices são numerados de $\emptyset$ a $V-1$

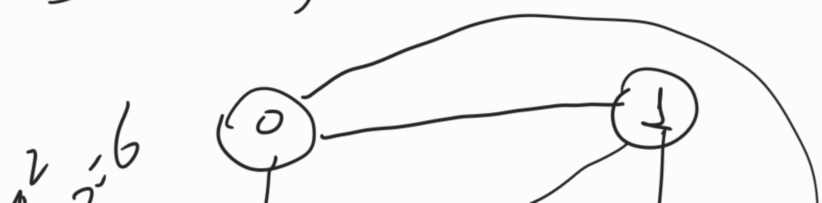


→ Grafos dirigidos

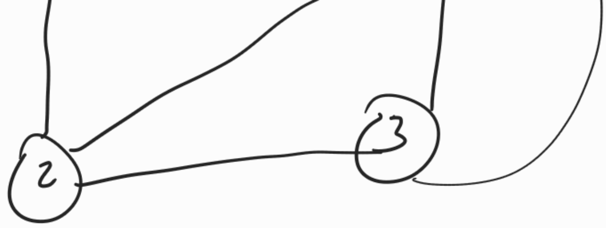


→ Grafo Completo

Um grafo com V vértices tem no máximo $V(V-1)$



$\frac{4\cdot 3}{2}$

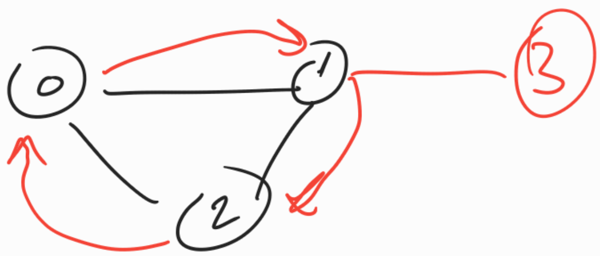


máximo $\frac{v(v-1)}{2}$ arestas

→ Caminho no grafo é uma sequência de vértices em que cada vértice sucessivo é adjacente ao predecessor no caminho.

↳ Caminho simples → os vértices e arestas são distintos

↳ Um ciclo é um caminho que é simples exceto pelo primeiro e último vértice que são os mesmos



→ Um grafo é conexo se há um caminho de cada vértice para todo outro vértice no grafo.

↳ Um grafo que não é conexo consiste de um conjunto de componentes conexos

→ Um grafo conexo acíclico também é chamado de ÁRVORE

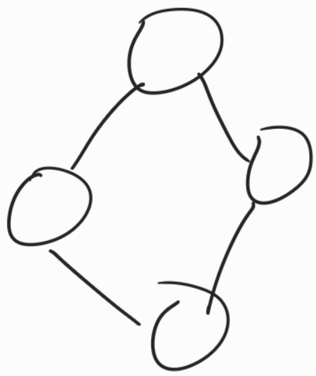
↳ Um conjunto de ÁRVORES é chamado de FLORESTA

→ ADT

```
typedef struct {int v; int w;} Edge;
Edge EDGE(int, int);
typedef struct graph* Graph;
Graph GRAPHinit(int);
```

Quantidade de vértices

```
void GRAPHInsertE(Graph, Edge);
void GRAPHRemoveE(Graph, Edge);
int GRAPHEdges(Edge[], Graph G);
Graph GRAPHCopy(Graph);
void GRAPHDestroy(Graph);
```



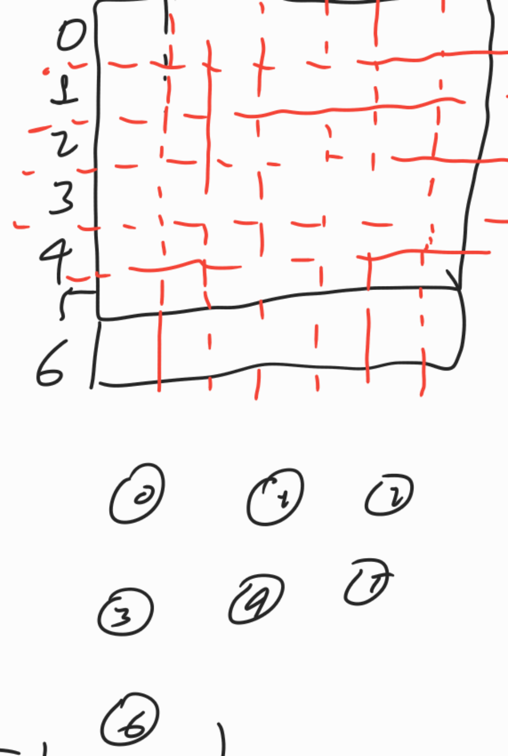
→ Matriz de Adjacência

4 + 4 + 8 = 16

```
struct graph {int V; int E; int **adj;};
```

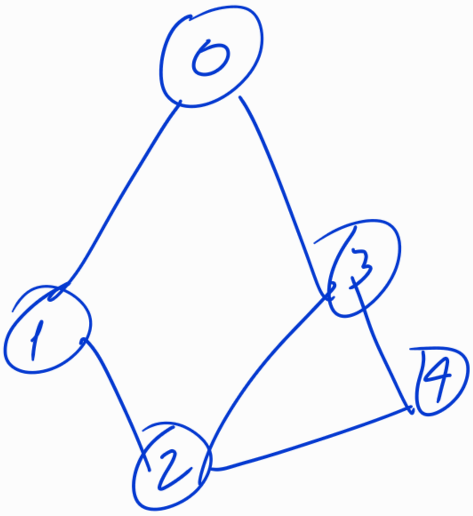
0 | 1 | 2 | 3 | 4 | 5 | 6

```
Graph GRAPHInit(int V)
{
    Graph G = malloc(sizeof(*G));
    G->V = V; G->E = 0;
    G->adj = MATRIXInit(V, V, 0);
    return G;
}
void GRAPHInsertE(Graph G, Edge e)
{
    int v = e.v, w = e.w;
    if (G->adj[v][w] == 0) G->E++;
    G->adj[v][w] = 1;
    G->adj[w][v] = 1;
}
void GRAPHRemoveE(Graph G, Edge e)
{
    int v = e.v, w = e.w;
    if (G->adj[v][w] == 1) G->E--;
    G->adj[v][w] = 0;
    G->adj[w][v] = 0;
}
```



```
int GRAPHEdges(Edge a[], Graph G)
{
  int v, w, E=0;
  for(v=0; v<G->V; v++)
    for(w = v+1; w<G->V; w++)
      if(G->adj[v][w] == 1)
        a[E++] = EDGE(v,w);
  return E;
}
```

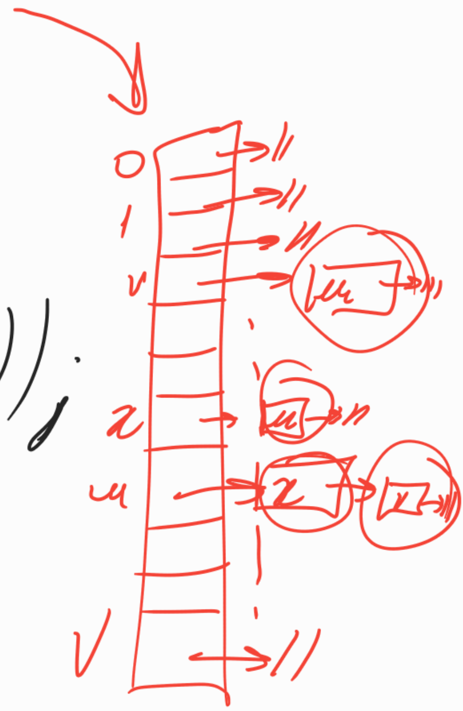
{0,1},{2,3},}



→ Lista de Adjacências

```
typedef struct node *link;
struct node{ int v; link next; };
struct graph{ int V,E;
              link *adj; };
```

```c
link NEW(int v, link next)
{
    link x = malloc(sizeof *x);
    if (x == NULL) tela_azul();
    x->v = v; x->next = next;
    return x;
}

Graph GRAPHInit(int V)
{
    int v;
    Graph G = malloc(sizeof(*G));
    G->V = V; G->E = 0;
    G->adj = malloc(V * sizeof(link));
    for (v = 0; v < V; v++)
        G->adj[v] = NULL;
    return G;
}

void GRAPHInsert(Graph G, Edge e)
{
    int v = e.v, w = e.w;
    G->adj[v] = NEW(w, G->adj[v]);
```



```
G->adj[u] = New(v, G->adj[u]);
G->E++;
}

int GRAPHEdges(Edge a[], Graph G)
{ int v, E=0; link t;
  for(v=0; v<G->V; v++)
    for(t=G->adj[v]; t!=NULL; t=t->next)
      if (v < t->v) a[E++] = EDGE(v, t->v);
  return E;
}
```

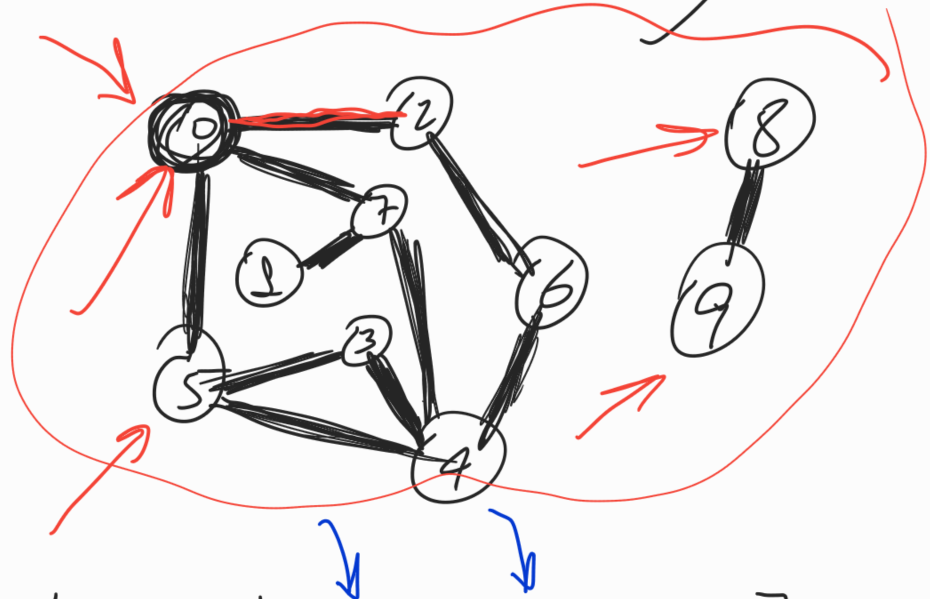
	Vetor de Arestas	Matriz Adj	Lista Adj
espaço	E	V^2	$V+E$
Inicializar	1	V^2	V
copy	E	V^2	E
destruir	1	V	E
Inserir E	1	1	1
Encontrar/Remover E	E	1	V

v é isolado? | E | V | 1

Caminho de u para v | E · lg V | V² | V+E

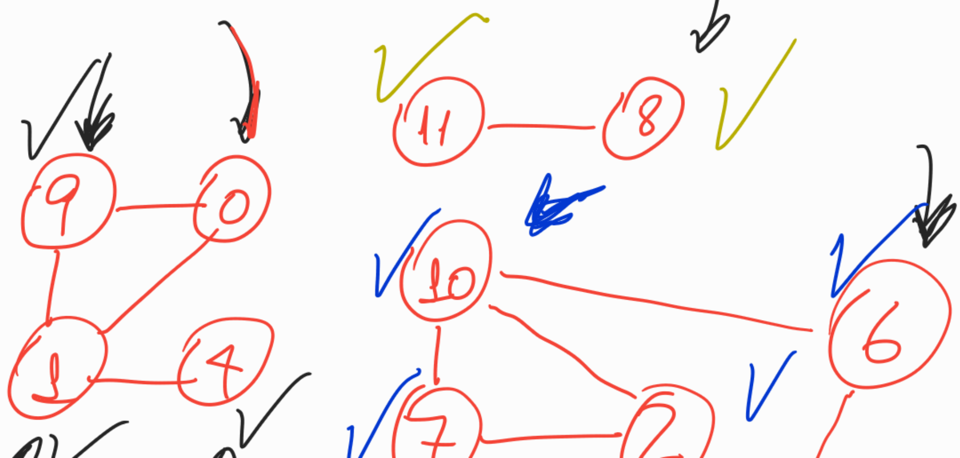
→ Busca em Profundidade (DFS)

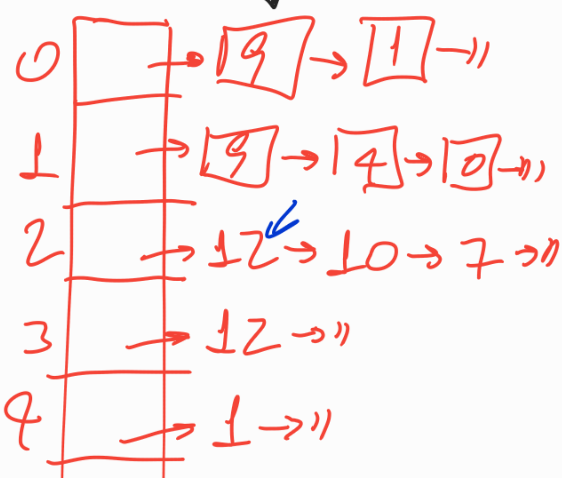
```
void dfsR(Graph G, Edge e)        Edge(v, w)
{
   int t, w = e.w;
   pre[w] = cnt++;
   for(t=0; t<G->V, t++)
      if (G->adj[w][t] != 0)
         if (pre[t] == -1)
            dfsR(G, EDGE(w, t));
}

int pre[V];
int main()
{
   INICIAGRAPH();
   for(int v=0; v<G->V; v++)
      pre[v] = -1;

   dfsR(G, EDGE(0, 0));
```

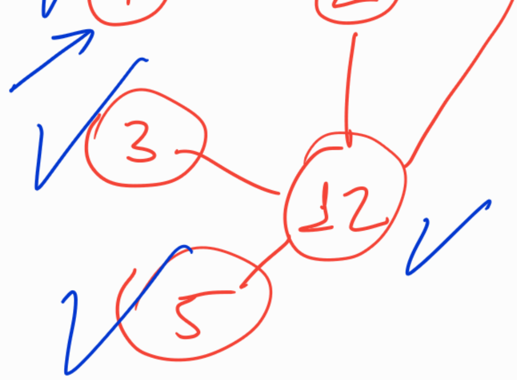


```
static int cnt, pre[maxV];
Void GRAPHSearch(Graph G)
{
 int v; int conexos = 0;
 cnt = 0;
 for (v=0; v<G->V; v++) pre[v] = -1;
 for(v=0; V<G->V; v++)
  if(pre[v] == -1) {
   dfsR(G, EDGE(v,v));
   conexos ++;
  }
}
```

13







5 → 12 → //
6 → 12 → 10 → //
7 → 10 → 2 → //
8 → 11 → //
9 → 1 → 0 → //
10 → 7 → 6 → 2 → //
11 → 8 → //
12 → 6 → 5 → 3 → 2 → //

pre

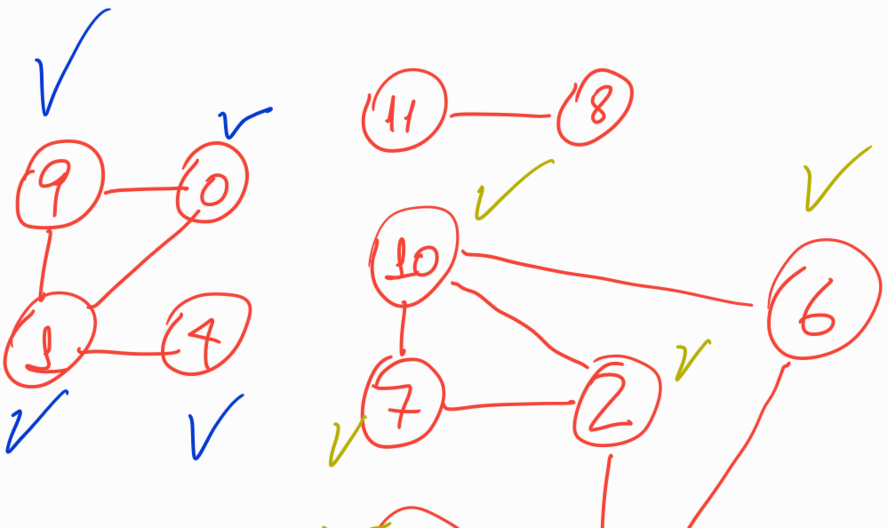
0	1	2	3	4	5	6	7	8	9	10	11	12
0	2	4	10	3	9	6	8	11	1	7	12	5

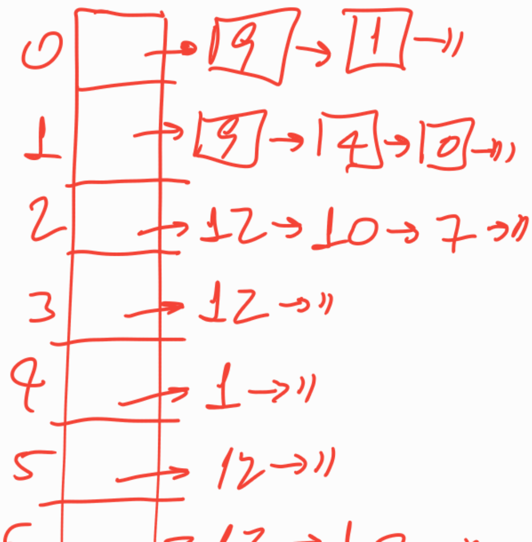
cnt = ~~0~~ ~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ ~~6~~ ~~7~~ ~~8~~ ~~9~~ ~~10~~ ~~11~~ ~~12~~ 13

Conexos = ~~0~~ ~~1~~ ~~2~~ 3

```
void dfsR(Graph G, Edge e)
{
  int t, w = e.w; link l;
  pre[w] = cnt++;
  for(l = G->adj[w]; l != NULL; l = l->next)
  { t = l->v;
    if (pre[t] == -1)
      dfsR(G, EDGE(w, t));
  }
}
```

```
Void bfs(Graph G, Edge e)        Edge ...
                                 {0,0}
{
  int v, w;
  Queueput(e); pre[e.w] = cnt++;
  while(!QUEUEempty())
  {
    e = QUEUEget();
    w = e.w;
    for(l = G->adj[w]; l != NULL; l = l->next)
    { t = l->v;
      if(pre[t] == -1){
        Queue put(EDGE(w, t));
        pre[t] = cnt++;
      }
  }
}
```





6 → 12 → 10 → //
7 → 10 → 2 → //
8 → 11 → //
9 → 1 → 0 → //
10 → 7 → 6 → 2 → //
11 → 8 → //
12 → 6 → 5 → 3 → 2 → //

pre

0	1	2	3	4	5	6	7	8	9	10	11	12
0	2	4	10	3	9	8	7	-1	1	6	-1	5

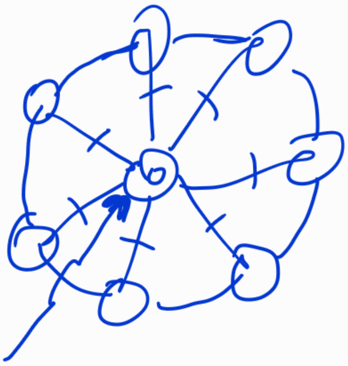
cnt = ~~0~~ ~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ ~~6~~ ~~7~~ ~~8~~ ~~9~~ ~~10~~ 11

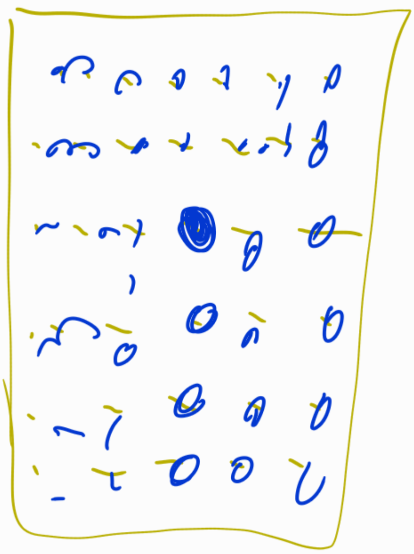
bfs {2, 12, 10, 7, 6, 5, 3}

fila [ ] → //

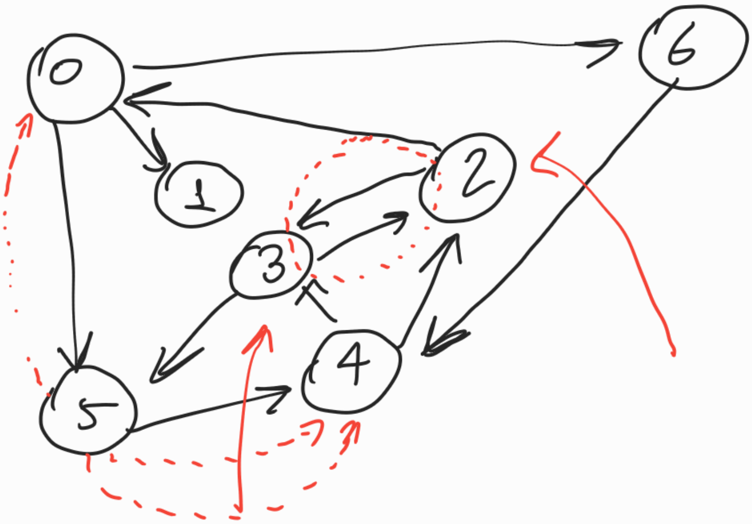
dfs {2, 12, 6, 10, 7, 5, 3}

e : {12, 3}





→ Grafo Dirigido (Digraphs)



0 → 6 → 5 → 1 → //
1 → //
2 → 3 → 0 → //
3 → 5 → 2 → //
4 → 3 → 2 → //
5 → 4 → //
6 → 4 → //
7
8

<#vertices>

0 6 7
0 1
0 5

0 6 1 5

→ Regras do jogo

: Grafo dirigido (ou digrafo) é um conjunto de vértices e um conjunto de arestas dirigidas que conectam um par de vértices (SEM ARESTAS DUPLICADAS).
Dizemos que uma aresta vai DE um primeiro vértice PARA o seu segundo vértice

primeiro

```
Edge { int v, w };
```

segundo

$v \to w$

: Caminho dirigido em um digrafo é uma lista de vértices no qual existe uma aresta dirigida conectando cada vértice da lista a seu sucessor. Dizemos de um vértice t é alcançável de um vértice s se existe um caminho dirigido de s a t.

Quantidade de arestas em um grafo não dirigido V^2, já em um grafo dirigido temos 2^{V^2}

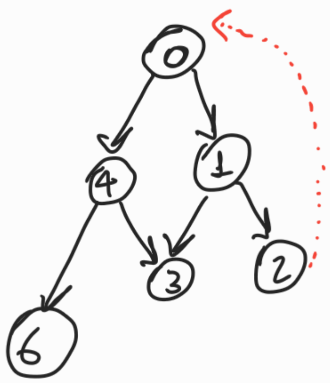
→ Como inverter as arestas de um grafo dirigido?

```
Graph GRAPHreverse(Graph G)
{
  int v; link t;
```

```
Graph R = GRAPHinit(G->V);
for (v=0; v<G->V; v++)
    for (t=G->adj[v]; t!=NULL; t=t->next)
        GRAPHinsert(R, EDGE(t->v, v));

return R;
}
```

: Grafo dirigido Acíclico (DAG - directed acyclic graph)

é um grafo dirigido que não possui ciclos.

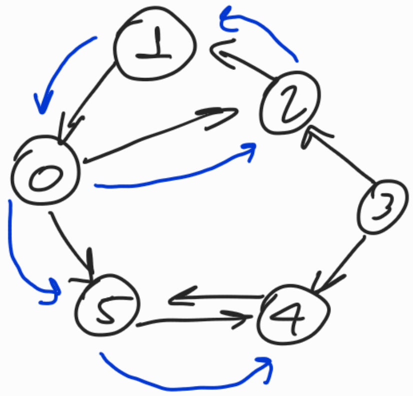


: Grafo dirigido fortemente conexo: se todos os vértices são alcançáveis a partir de todos os vértices

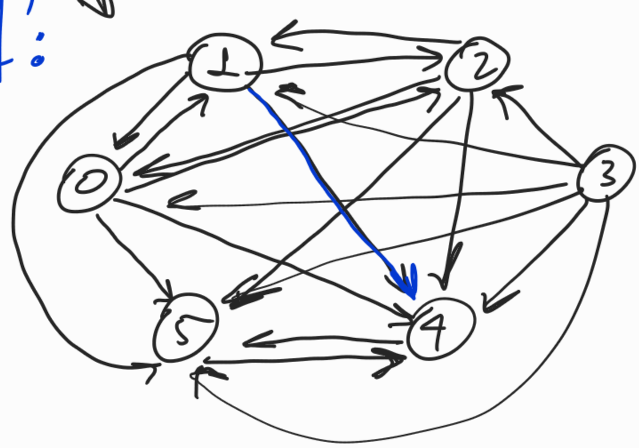
→ Alcançabilidade e Fecho Transitivo

Fecho transitivo de um grafo dirigido é um grafo dirigido com o mesmo conjunto de vértices mas com uma aresta de s a t no fecho transitivo se e somente se existe um caminho dirigido de s a t no grafo dirigido.

A:



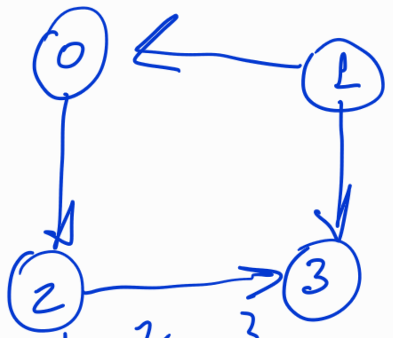
A':



→ Floyd Warshall

```
Void GRAPHtc(Graph G)
{
  int i, s, t;
  G->tc = MATRIXint(G->V, G->V, 0);
  for(s=0; s<G->V; s++)
    for(t=0; t<G->V; t++)
      G->tc[s][t] = G->adj[s][t];
  for(s=0; s<G->V; s++) G->tc[s][s] = 1;

  for(i=0; i<G->V; i++)
    for(s=0; s<G->V; s++)
      if(G->tc[s][i] == 1)
        for(t=0; t<G->V; t++)
          if(G->tc[i][t] == 1)
            G->tc[s][t] = 1;
}

int GRAPHreach(Graph G, int s, int t)
{
  return G->tc[s][t];
}
```

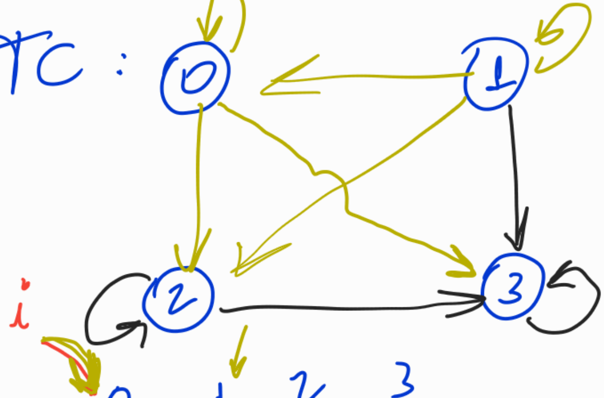
A:



Adj:

	0	1	2	3
0	0	1	0	0
1	0	0	0	0
2	1	0	0	0
3	0	1	1	0

TC:

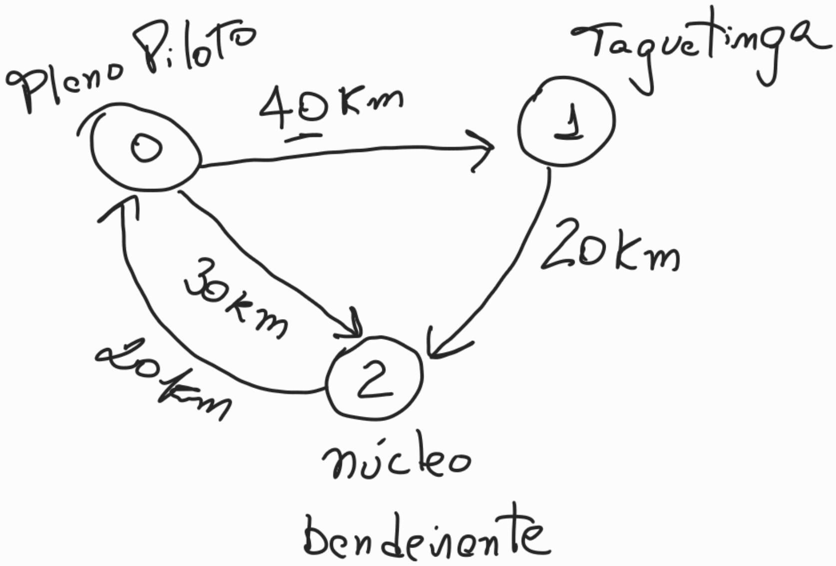


i: 2
j: 1
t:

tc:

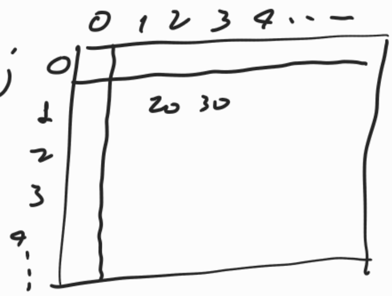
	0	1	2	3
0	1	1	0	0
1	0	1	0	0
2	1	1	1	0
3	1	1	1	1

→ Peso nas arestas



→ Como Implementar o peso na aresta?

→ Matriz Adj



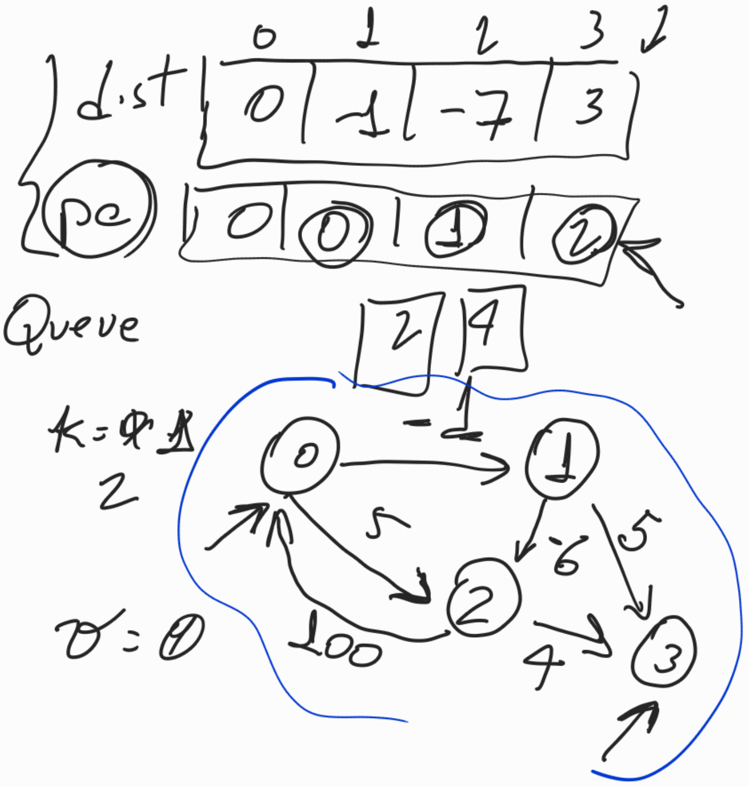
→ Lista Adj

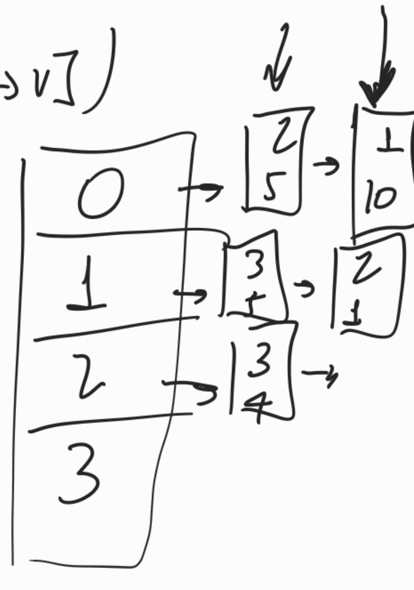
struct Edge { int v, int c, link next;

→ Bellman Ford)

→ Dijkstra)

```
bool GRAPHcptBF(Graph G, int s, int *pa, int *dist)
{ bool onqueue[1000];
  for(int v = 0; v < G->V; v++)
      pa[v]=-1, dist[v]=INT_MAX, onqueue[v]=false;
  pa[s]=s; dist[s]=0;
  Queueinit(G->E);
  Queueput(s);
  onqueue[s]=true;
  Queueput(V); int k=0;
  while(1)
  {
     int v = Queueget();
     if (v<V)
     {
       for (link a= G->adj[v]; a!=NULL; a=a->next)
       {
          if(dist[v] + a->c < dist[a->v])
          { dist[a->v] = dist[v]+a->c;
            pa[a->v] = v;
            if(onqueue[e->v]==false)
            {
               Queueput(a->v);
               onqueue[e->v]=true;
```





```
            onfueue
        }
    }
  }
else
{   if(Queueempty()) return true;
    if(++k >= G->V) return false;
    QueueputT(V)
    for(int t=0; t<G->V; t++)
        onqueue[t]=false;
  }
 }
}
```

→ Dijkstra (ingênuo)

↳ Caminhos mínimos

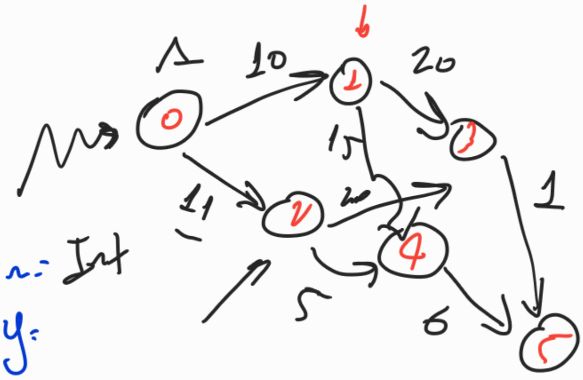
↳ Somente pesos positivos!

```
void GRAPHcptD1(Graph G, int s, int *pa,
                int *dist)
{
bool mature[1000];
```

```
for(int v=0; v<G->V; v++)
    pa[v] = -1, mature[v]= false, dist[v]=INT_MAX;

pa[s] = s; dist[s]=0;
while (true)
{
  int min =INT_MAX;
  int y;
  for(int z =0; z<G->V; z++)
  {
      if(mature[z]) continue;
      if(dist[z] <min)
           min = dist[z], y=z;
  }
  if(min == INT_MAX) break;
  for(link a = G->adj[y]; a!=NULL; a =a->next)
  {
      if(mature[a->v]) continue;
      if(dist[y]+ a->c < dist[a->v])
      {  dist[a->v] = dist[y]+a->c;
         pa[a->v] = y;
      }
  }
  mature[y] =true;
}
```

3
1
0
0

min: Int
y:



	0	1	2	3	4	5
pa	0	0	0	1	2	4
dist	0	10	11	30	16	22
mature	1	1	1	1	1	1